

FAQ

Q What is Microsoft's Common Language Runtime (CLR) used for?

A This is the core of Microsoft's .NET system. It is a virtual machine similar to the Java Virtual Machine. A virtual machine simulates a computer that executes specific machine code – in this case, Microsoft Intermediate Language (MSIL).

When you write a program using any .NET language, the compiler converts what you write into MSIL. When the program is run, the MSIL is presented to the CLR, which partly converts it into the native machine code of the processor executing it.

In theory, this allows an application to be written and compiled once, and run on any computer regardless of its OS. The CLR is available only for PCs running Windows and some mobile devices.

Mike James

IT author, lecturer
and programmer

mike@computershopper.co.uk

Check a server is running

Do you have an important server that you need to be available at all times? Mike James shows you how to write a program that can check its status automatically



If you've got a server on the internet, such as one running an important website, you don't want it to be unavailable. Still, problems do happen, so it's best to take a proactive approach and be warned as soon as it goes down. This way, as soon as there's a problem you can get it fixed, and you will suffer less downtime as a result.

It isn't difficult to write an application to scan a list of internet servers to see if they are available. Taking the DIY approach has its advantages, too, as you can customise your application to do exactly what you want, such as sending an email should a website become unavailable. In this month's project, we are going to write an application we'll call SiteScan. This is a simple C# application that keeps track of when a list of servers is contactable via the internet. If the application can't contact a server, it could be a problem with your internet connection. Even so, this will still alert you to a problem, which you can fix instantly.

To make the program work, we're going to get our application to send each of the servers in the list a ping request and collect data on whether they respond or not. If you're interested in testing a specific protocol such as HTTP for a web server, you will have to replace the ping class with custom code that checks to see if that service is working.

INITIALISING THE GRID

For this project, you'll need .NET 2.0, as this was the first version to introduce the ping class. Start a new Windows Forms application using C#. Next, we need something to hold the list of sites to be scanned. There are several ways of doing this, including a Listbox or

Textbox. If we are going to use the same component to display the results of the scan, the obvious choice is the DataGridView, though it can be more complicated to use.

Place an instance of the DataGridView on the form and a button labelled StartScan. Customise the DataGridView using its Task Editor, which can be accessed by clicking on the small arrow that appears at the top when the button is selected. Add three columns, labelled Address, Status and Time, and appropriate column header text.

APPLYING THE INITIALISATION

In a full application, you would need to add some simple code to allow the user to save a list of servers to test. For now, it's simpler to hard-code the list as part of an initialisation method. The user can still add extra servers by typing URLs or IP addresses directly into the grid, but there is no way of saving them for reuse. The initialisation can be included in the form's constructor:

```
public Form1()
{
    InitializeComponent();
    dataGridView1.Rows.Clear();
    dataGridView1.Rows.Add();
    dataGridView1.Rows[0].
        Cells["Address"].Value = "first URL";
    dataGridView1.Rows[0].
        Cells["Status"].Value = "Not Tested";
    dataGridView1.Rows.Add();
    dataGridView1.Rows[1].
        Cells["Address"].Value = "IP
```

```

        address";
dataGridView1.Rows[1].Cells["Status"].Value = "Not Tested";
}

```

You can add as many URLs and IP addresses as you like using similar instructions. While you can select which column of the grid is used via an index, as in Cells[0] for the first column, using the column name makes the code more readable.

USING THE TIMER COMPONENT

We are going to have to arrange to ping each server at a set interval. There are many ways of implementing this, but the simplest is to use a Timer component. Place a Timer on the form and add the following code to the start of the Form definition:

```
const int interval = 5;
```

This sets the scanning interval to five minutes, which should be sufficient. Pinging a server too often can constitute an attack and you could get into trouble, and it could also degrade the server's performance. In principle, you should get permission to ping a public server on a regular basis. However, pinging a server once every five or 10 minutes should have no detectable impact on its performance.

To implement the timer interval, we need to add to the end of the form's constructor:

```

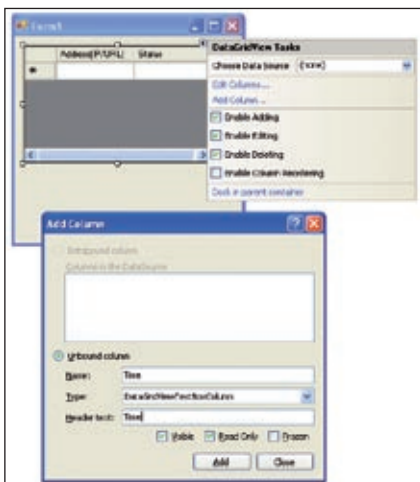
timer1.Enabled = false;
timer1.Interval = interval * 60 * 1000;
}

```

This disables the timer and sets its interval to five minutes. The timer works in milliseconds, so we multiply our interval variable by 60 to convert to seconds, and then 1,000 to give milliseconds. While you are testing the program you might want to decrease the testing interval, but keep in mind the comments about pinging a server too often.

The StartScan button simply has to enable the timer, but rather than wait for the entire interval before scanning, it's a good idea to call the timer's event handler directly first:

```
private void button1_Click(object sender, EventArgs e)
```



▲ Create the SiteScan form

```

{
    timer1_Tick(timer1, new EventArgs());
    timer1.Enabled = true;
}

```

SETTING UP THE APPLICATION

Now we have the basic structure needed to set up the application and start the timer running. Everything that the application does from this point on is concentrated in the timer routine.

It's a good idea to give the user feedback about what is going on, so add a Textbox to the form and set it to multi-line with a single vertical scrollbar. We can now post a message each time the timer event handler is called:

```

private void timer1_Tick(object sender, EventArgs e)
{
    textBox1.Text += "Update at:" +
        DateTime.Now.ToString() +
        Environment.NewLine;
    textBox1.Refresh();
}

```

If you run the program as it stands, you'll see the 'Update at' message appear at the set interval.

For each of the rows of the grid that contain a valid address, we have to ping the server. The simplest way of doing this is to use a 'foreach' loop, and so the timer tick method continues:

```

foreach (DataGridViewRow row in dataGridView1.Rows)
{
    if (row.Cells["Address"].Value != null)
    {

```

We are ready to ping the server. To do this, we assume there is a method that will accept the address, a retry and pause specification. The retry is the number of times to ping in the case of a failure. The pause is the time in seconds to leave between each attempt. Some servers go to sleep if they aren't being used, so an initial ping might fail for this reason. However, there isn't much point trying to ping the server immediately after it has failed to respond, because it needs time to wake up, hence the need for a pause between pings. In this case, we are providing a single value for the retry and pause to be used with all the servers in the list. If you want to be more flexible, you could store values to suit each server to be tested in additional columns in the grid.

Next we perform the ping:

```

PingReply reply = DoPing(
    row.Cells["Address"].Value.
    ToString(),
    retry, pause);

```

We will write the DoPing method later. It could easily be replaced by a DoWebPage method if you wanted to test to see if a webpage were available, or a more general DoProtocol method.

The DoPing method returns a PingReply object, which contains the result of the ping. To use the ping facilities that .NET 2.0 provides, we have to add this line to the start of the program:

```
using System.Net.NetworkInformation;
```

Now we can transfer the Status and Time property in the PingReply object to the grid to

provide the user with feedback on what has happened:

```

row.Cells["Status"].Value =
    reply.Status.ToString();
row.Cells["Time"].Value =
    reply.RoundtripTime.
    ToString();
dataGridView1.Refresh();
}
}

```

All that remains is to create the DoPing method. First we need to create a suitable Ping object and PingReply variable:

```

private PingReply DoPing(string address, int retry, int pause)
{
    Ping pinger = new Ping();
    PingReply reply=null;

```

We now try to ping the number of times specified by retry:

```

for (int i = 1; i <= retry; i++)
{
    reply= pinger.Send(address);

```

The Send method comes in different forms and there is also an asynchronous send method, which returns immediately. The 'blocking' Send is easier to use, but it halts the execution of the program until the server replies or times out. You can set the timeout to something other than its default five seconds using another version of the Send method. If the reply is successful, we just bring the for loop to an end:

```

if (reply.Status == IPStatus.
    Success) break;

```

Otherwise, we wait for the specified pause time and try again, or simply end the loop and return the result:

```

    Thread.Sleep(pause*1000);
}
return reply;
}

```

Notice the way the pause is implemented by putting the current thread to sleep. This gives other processes the opportunity to run. To use the Thread object, we need to add this to the start of the program:

```
using System.Threading;
```

If you run the program and click the StartScan button, you should see the status of all the sites listed change at the update interval. At this point, it is worth adding another button labelled StopScan with event handler:

```

private void button2_Click(object sender, EventArgs e)
{
    timer1.Enabled = false;
}

```

The user interface should now look something like the one shown on the following page.

O'REILLY

Programming Collective Intelligence

★★★★★

£16

From www.amazon.co.uk

Collective intelligence is the idea that small simple algorithms can produce powerful programs if they communicate with each other and make collective decisions. You see collective intelligence in action when birds flock and wheel overhead, and ants, termites and other insects build incredible structures and solve complex problems.

Toby Segaran's book isn't really about collective intelligence, but about using artificial intelligence in Python programs. Python might not be the most obvious choice of computer language for artificial intelligence, but Segaran's code is easy to read even if you don't use this language.

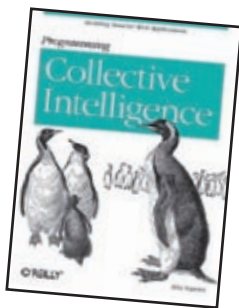
The book's topics include clustering, decision trees, Bayesian filtering and more. The explanations are down to earth and practical, and the code very easy to follow. The book's approach is so practical that you can take the code and try to use it in one of your own applications. However, if you do, you'll discover that real life is never as easy as the book makes out, and you need to put a lot of work into getting a full working application. Therein lies one of the book's biggest but most subtle problems: it is very light on theory. Without the theory behind how the algorithms work, it's hard to know how to fix a practical problem or even what's causing it.

This is an excellent introduction to intelligent method in programming, but it raises as many practical problems as it solves. You'll need to read additional books if you're going to program intelligent systems.

SUMMARY

- ☒ Practical approach
- ☒ Light on theory

ISBN 0596529325
PAGES 360



ADDISON WESLEY

Exploiting Online Games: Cheating Massively Distributed Systems

★★★★★

£19

From www.amazon.co.uk

There are now hundreds of online games and millions of players. By manipulating, changing and tweaking the code that makes these games, it's possible to cheat at them. Greg Hoglund and Gary McGraw's book looks at how this can be done. It deals with the specifics of the big online games, World of Warcraft and Second Life, but this is more than a book about doing better in games.

The theories and ideas behind it are a very interesting read if you're interested in hacking and security in general, as the techniques described could be used to attack any distributed system. Most attacks described are well known and you wouldn't get very far if you attempted to use them. Their real value is to show you have to think to create new attacks.

The point is that knowing how a distributed system might be attacked gives you a clearer idea of what you have to do to protect it. Systems like these distributed gaming applications are the way of the future. Software has to become increasingly distributed and parallel, because we are moving from an era where the power of a single processor increases without limit to one where the number of processors increases. The attacks currently being made on gaming systems probably represent the sort of security onslaughts to which general future systems will be subjected.

You will need to be able to read C and fight through a couple of over-long sections to get through this book, but persevere with it and you can't help but learn something.

SUMMARY

- ☒ Thought-provoking
- ☒ Some sections hard to read

ISBN 0132271915
PAGES 384



There is one point to keep in mind when using a Timer component to generate regular events. A Timer component uses the same thread of execution as the user interface. So, the form the Timer component is placed on updates itself. This makes programming simple, because the timer Tick event handler can access the components on the form without any issues to worry about. There are alternative timers that use their own thread of execution; these are more accurate and flexible, but more difficult to use in conjunction with a form. The biggest problem with using a Timer component is that while the Tick event handler is executing, the user interface provided by the form is frozen. In the case of our application, the Tick event doesn't take long to process and the freeze shouldn't cause problems. If the freeze becomes unacceptable, you can either switch to using one of the other Timer objects that .NET offers, or use this line:

```
Application.DoEvents();
```

Place this line at any point in the Tick event handler to allow the user interface to run.

GATHERING THE DATA

The program as developed so far is fine for a real-time view of the status of a list of servers, but useless if you want statistics on availability. The solution is to add a recording facility that writes the data to disk. There are two difficulties to overcome: first, minimising the volume of data and second, finding a good location for the data. We could just write the status to disk every time the availability of the server is checked, but this would generate more data than necessary. All we need is a record of the time that a server's status changes, from online to offline and from offline to online. This is very easy to arrange. We just have to trigger a write on a change of status.

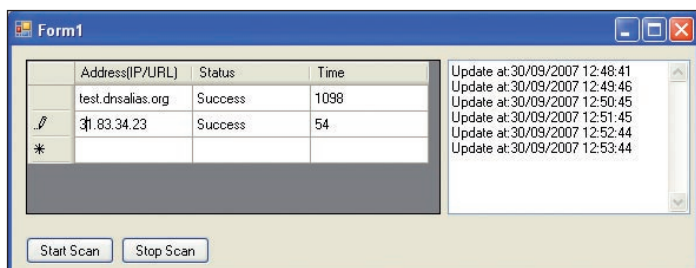
The problem of where to write the data is more difficult. You could create a special file to write the data to, but custom files tend to be messy, and the user has the problem of trying to

find them when they lose the documentation you provide. A better solution is to store such small amounts of transient data in a system-provided location, such as the Log File. This has the advantage that the user can inspect the results using the Event Viewer and can manage the custom log in the same way as any other log. It also has the advantage that, for the right sort of data, it's very easy to use.

USING THE EVENTLOG

.NET provides an EventLog component. This allows you to create a custom EventLog on a specified machine. The only complication is that you have to register the source of the events that the EventLog will handle, and once you have registered that source it can send events only to that particular EventLog. To create a custom event log, all we have to add to the end of the form's constructor are these two lines:

```
eventLog1.Log = "SiteStatus";
eventLog1.Source = "SiteScan";
```



◀ The user interface

This either opens an existing log called SiteStatus or creates a new one if it doesn't already exist. It then registers SiteScan as a source of events. You can register any name as a source and it doesn't have to be the name of the program writing to the log. You can also register multiple sources with a log, and you can register custom sources in the existing standard system log. However, it seems simpler from the user's point of view to group all the custom event messages into their

own log. If you accidentally create the wrong log, or you no longer need it, you can remove it by using:

```
if (System.Diagnostics.EventLog.
    Exists("MyCustomLog"))
{
    System.Diagnostics.EventLog.
        Delete("MyCustomLog");
}
```

Similarly, if you accidentally register, or you no longer want, a source associated with a log, then you can remove it by using:

```
if (System.Diagnostics.EventLog.
    SourceExists("Source1"))
{
    System.Diagnostics.EventLog.
        DeleteEventSource("Source1");
}
```

Now we have created the event log, we can write some data to it. Place the following code immediately following the DoPing instruction:

```
if (row.Cells["Status"].Value.
    ToString() != reply.Status.
    ToString())
{
    eventLog1.WriteEntry(
        row.Cells["Address"].Value.
        ToString() +
        "," + reply.Status.ToString() +
        "," + reply.RoundtripTime.
        ToString());
}
```

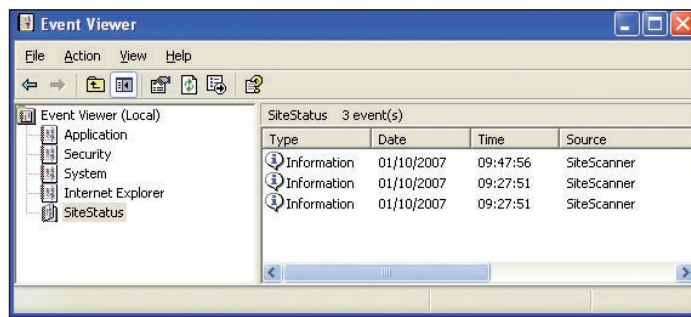
If the status of the server has changed since the last time it was checked, an event consisting of the server address, status and roundtrip time is written to the log.

TROUBLESHOOTING

There is one remaining problem. Suppose a server has been on since SiteScan started work. In the log, there is just a single record reporting its state as 'on' at the time SiteScan started. If you try to process the data to work out the percentage of time the server has been on, you can't know if SiteScan is still running, or if it stopped some days ago. This makes a difference to what you think the total time monitored will be. There are several solutions, but the simplest is to write a final event record when SiteScan is closed. To mark this as a final event, we can set the event type code to one instead of the default value of zero.

Change the Finish Scan button's event handler so it reads:

```
private void button2_Click(
    object sender, EventArgs e)
{
    timer1.Enabled = false;
    foreach (DataGridViewRow row
        in dataGridView1.Rows)
    {
        if (row.Cells["Address"].Value !=
            null)
        {
            eventLog1.WriteEntry(
                row.Cells["Address"].Value.
                ToString() +
                "," + row.Cells["Status"].
```



◀ Viewing the log file via the Event Viewer

```
Value.ToString()+
    "," + row.Cells["Time"].Value.
    ToString(),
    EventLogEntryType.Information,
    1);
}
```

This writes a final event record to the log when the scan is stopped. To make this work, we have to add this line to the start of the program:

```
using System.Diagnostics;
```

Now we have two types of event record. Event type zero indicates that a server has changed state, and an event type one indicates that the monitoring has stopped.

If you now run the program, you'll find that a new log file has been created and events are written to it when a server connects or disconnects. You can view the log file via the Event Viewer, which you can find in the Control Panel under Administrative Tools. This also allows you to save a copy of the log file, clear it and set up custom views.

ANALYSING THE LOG

There are several ways to analyse the log to create a report of server performance. From the Event Viewer, you could save the log in Comma Separated Values (CSV), which is easy to process in Excel. Writing a custom application to process the data is also easy with the help of the EventLog component. For example, to compute the time a specified server is available, you first need to start a new Windows Forms project and place an instance of the Event Viewer component on it. In this simple example, all the computation is performed in the OnLoad event handler.

We start by defining all the variables to hold the on and off times and the status of the server:

```
private void Form1_Load(object
    sender, EventArgs e)
{
    eventLog1.Log = "SiteStatus";
    DateTime OnTime=new DateTime(0);
    DateTime OffTime = new
        DateTime(0);
    TimeSpan TotalTime=new TimeSpan();
    Boolean On;
    Boolean StartOn=false;
```

Next we open the log:

```
eventLog1.Log = "SiteStatus";
```

Reading the log is simply a matter of stepping through its Entries collection one entry at a time:

```
foreach (EventLogEntry entry
    in eventLog1.Entries)
{
    string[] data = entry.Message.
        Split(',');
    string Address = data[0];
    string Status=data[1];
    string RoundTripTime = data[2];
```

As the message is comma-delimited, we can use the Split method to chop it up into Address, Status and RoundTripTime.

We now check to see if this is the server on which we need the statistics. In your program, TargetServer should be replaced by the URL or IP address of the server:

```
if(Address==TargetServer)
{
    On=(Status=="Success");
```

The variable On is now true if the server is connected and false otherwise.

EXPLORING THE POSSIBILITIES

There are now three possibilities for us to consider. This could be the first On entry we have encountered, in which case we need to store the time it first came on:

```
if(!StartOn & On)
{
    StartOn=true;
    OnTime=entry.TimeGenerated;
}
```

It could be the first Off entry we have encountered, in which case we can calculate the time interval it has been on:

```
if(StartOn & !On)
{
    StartOn=false;
    OffTime=entry.TimeGenerated;
    TotalTime+=OffTime-OnTime;
}
```

Notice the use of a TimeSpan object to store the total time.

The third situation is where multiple on or off entries are encountered in the sequence. To calculate the interval correctly, we will just need the first on time, so we can ignore any subsequent on records. We also can ignore any subsequent off entries, and so this completes the analysis. The program doesn't deal with final entries that signify that the scan has come to an end. It will give the wrong answer if the server never goes off, because we will just have an on time and no off time to subtract. Both these problems will be easy to fix by adding extra conditions. **CS**